3. The Basics of Dynamic Web Pages

# Your First View: Dynamic Content

In Django, a view is responsible for processing a web request and returning a web response. Here's a simple example of a view that renders dynamic content:

```python
from django.http import HttpResponse

def hello_world(request):
    return HttpResponse("Hello, World!")
```

# Mapping URLs to Views

To connect a URL to a view, you use the `urls.py` file in your Django app. For example:

```python
from django.urls import path
from .views import hello_world

urlpatterns = [
    path('hello/', hello_world, name='hello_world'),
]
```

This maps the URL `/hello/` to the `hello_world` view.

# How Django Processes a Request

1. Django receives a request.
2. The URL patterns are examined to determine which view to call.
3. The view processes the request and returns an HTTP response.

# URL Configurations and Loose Coupling

Django encourages loose coupling by separating URL configuration from views. This improves code organization and maintainability.

# 404 Errors

If a URL doesn't match any pattern, Django raises a 404 error. You can customize the 404 page by creating a `404.html` template.

# Your Second View: Dynamic URLs

Views can accept parameters from the URL. For example:

```python
from django.http import HttpResponse

def greet_user(request, username):
    return HttpResponse(f"Hello, {username}!")
```

URL configuration:

```python
from django.urls import path
from .views import greet_user

urlpatterns = [
    path('greet/<str:username>/', greet_user, name='greet_user'),
]
```

# A Word About Pretty URLs

Django encourages the use of human-readable and SEO-friendly URLs.

# Wildcard URL Patterns

Wildcard patterns capture parts of the URL. Example:

```python
path('articles/<int:article_id>/', view_article, name='view_article')
```

4. The Django Template System

# Template System Basics

Django's template system allows you to define HTML templates with embedded Python-like syntax.

# Using the Template System

Example template (`hello.html`):

```html
<!DOCTYPE html>
<html>
<head>
    <title>Greetings</title>
</head>
<body>
    <h1>Hello, {{ user }}!</h1>
```

</body>
</html>

# Creating Template Objects

In a view, you render the template and provide a context:

```python
from django.shortcuts import render

def hello_user(request, username):
    return render(request, 'hello.html', {'user': username})
```

# Rendering a Template

Use the `render()` function to render a template with a given context.

# Multiple Contexts, Same Template

You can pass multiple contexts to a template:

```python
context1 = {'variable1': 'value1'}
context2 = {'variable2': 'value2'}

return render(request, 'example.html', {**context1, **context2})
```

# Context Variable Lookup

Access variables in the template using double curly braces: `{{ variable }}`.

# Playing with Context Objects

```python
context = {'numbers': [1, 2, 3]}
```

In the template:

```html
{% for number in numbers %}
    {{ number }}
{% endfor %}
```

# Basic Template Tags and Filters

Django provides tags and filters to perform logic and modify displayed content in templates.

# Using Templates in Views

Use the `render_to_response()` shortcut function:

```python
from django.shortcuts import render_to_response

def my_view(request):
    return render_to_response('template_name.html', {'variable': 'value'})
```

# Template Loading

Django looks for templates in the `templates` directory of each app. You can organize templates into subdirectories.

# The `locals()` Trick

Pass all local variables to the template using `locals()`:

```python
return render(request, 'template_name.html', locals())
```

# The `include` Template Tag

Include other templates using the `{% include %}` tag:

```html
{% include 'header.html' %}
```

# Template Inheritance

Create a base template with common elements:

```html
<!-- base.html -->
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}Default Title{% endblock %}</title>
</head>
<body>
    <div id="content">
        {% block content %}{% endblock %}
    </div>
</body>
</html>
```

Inherit from the base template:

```html
<!-- child.html -->
{% extends 'base.html' %}

{% block title %}Custom Title{% endblock %}

{% block content %}
    <p>This is the child template content.</p>
{% endblock %}
```

This allows you to reuse and extend templates in a modular way.